C++ Class Library Data Management for Scientific Visualization

Al Globus, Computer Sciences Corporation[1]
globus@nas.nasa.gov

Abstract

Scientific visualization strives to convert large data sets into comprehensible pictures. Data sets generated by modern instruments and super-computers are so large that visualization is often crucial to comprehension. Visualization involves a great deal of numerical programming: vector arithmetic, inverting matrices, solving ODE's, etc. Commercial C++ class libraries implement most of what's needed, but have a fatal memory management performance flaw: data to be operated on is always copied from memory or disk into newly allocated space. On large data sets, these copies kill performance. A small class hierarchy has been implemented to address this problem. The base class implements reference count memory management and contains a virtual destructor, but does no allocation or deallocation. Derived classes are added to implement different ways of allocating memory and importing data into the system. This solution has been tested in our own C++ numerical class library -- the library we wish we didn't have to write.

I. Introduction

Data sets of current interest at our facility range in size from 600 megabytes to 160 *giga*bytes [Globus92]. To use a commercial C++ numerical class library for visualization purposes, our data must be passed to the library, typically by a constructor. Vector, matrix, and array class constructors in existing libraries either copy a memory buffer or read from a disk file [Dyad92,RW92]. Copies are expensive in physical memory and disk IO is expensive in time. Unnecessary copies and/or disk IO are unacceptable for our work.

Constructors are needed that can incorporate large blocks of data into an object without making a copy, and destructors are needed that delete the data when it is no longer needed. Allocation and deallocation must be done by executing *application specific code*. This can be done with a class hierarchy based at the class `tData`[2]. `tData` implements reference counting memory management and has a virtual destructor. Application programmers write derived classes implementing constructors and destructors that import (or allocate) and delete application specific data as desired.

In reference count memory management, each block of data is an object that keeps a count of the

1. This work is supported through NASA contract NAS2-12961.
2. In this paper, class names start with ‘t’, a habit picked up from MacApp. Also, code fragments will be in the `courier` font.

number of pointers referencing the block. When the reference count goes to 0, the data block is deallocated. Note that reference count memory management defeats normal C++ memory management of local variables and can cause problems. Reference count memory management is not central to the approach advocated by this paper. It is used here because the implementation is very simple and will not distract from our main point.

Note that if data initialized by the application is imported into a numerical library, the way the library expects data to be laid out in memory must be documented. E.g., for multi-dimensional arrays, the index increasing fastest with increasing memory address must be known.

II. Problem

A description of the `tData` class hierarchy follows, but first, consider two situations that arise in scientific visualization programming where the `tData` memory management approach vastly improves performance over that achievable using current commercial C++ numerical class libraries. The performance improvements derive solely from eliminating memory copies and disk IO.

1. Extensible visualization systems such as AVS [Upson89], SGI Explorer [SGI92a,SGI92b], and FAST [Bancroft90] are customized by writing ‘modules.’ These modules are separate UNIX processes that communicate locally via shared memory and to remote machines via sockets. This communication is handled by the visualization system. Module programmers need only write functions that input and output visualization system data types. Thus, a useful C++ numerical library must operate on data allocated by the visualization system, not the C++ library.

2. Some important visualization techniques access a very small, but unpredictable, portion of a data set. If data is on disk, the access pattern’s unpredictability prevents reading the right portion *a priori*. However, one can memory map the file [UNIX]. A memory mapped file is added to a process’ virtual memory space without actually reading the file into physical memory. The virtual memory system will transparently read only those portions of the data actually referenced by subsequent code; i.e., physical memory acts as a cache for data on disk. Thus, code which accesses memory mapped files or in-core buffers is identical, although performance will vary.

   Memory mapping can reduce IO by orders of magnitude. Consider following the path of a set of particles through a time dependent 3D vector field. Conceptually, this is the equivalent of releasing massless particles into a flow to see where they go. Mathematically,

   given the Eulerian vector field: $\vec{v}(\vec{x}, t)$

   find the particle path of $\vec{X}$ by solving: $\frac{d\vec{x}}{dt} = \vec{v}(\vec{x}, t)$

   with $\vec{x} = \vec{X}$ at $t = t_0$

   In our work, $\vec{v}(\vec{x}, t)$ is usually sampled on a curvilinear 3D grid in space and regularly in time. The amount of the data that must be accessed to solve $\frac{d\vec{x}}{dt} = \vec{v}(\vec{x}, t)$ is O(number-of- particles). The size of $\vec{v}(\vec{x}, t)$ is O(number-of-grid-points). The number of particles per time step is typically ten thousand or so [Levit92], whereas the number of sample points per time step is frequently in the millions [Globus92]. Eight sample points are necessary to interpolate values for one particle. Thus, in the unlikely worst case that no two particles access the same sample points, following 10,000 particles in a 3,000,000 point spatial grid requires access to 80,000 sample points per time step, or less than 2.7% of the data. Of course, the virtual memory system will access data not actually needed since disk reads have a minimum size, but the IO savings are still substantial. In one case, the time to calculate the motion of 100 particles for one step in a 2.8 million point data set went from many minutes to four seconds per time step [Globus93].

A useful C++ numerical library needs to operate on memory mapped data, but the present packages cannot.

We have seen two situations where a numerical package must operate on memory not allocated by the package. All such situations cannot be anticipated, so a flexible mechanism is needed to import (or allocate) and deallocate data from different sources.

III. A Solution

To operate on different kids of data transparently, a numerical class library need only differ in the way the data is imported and deallocated. C++ derived classes and virtual destructors can provide a such a mechanism. In our C++ numerical class library, we implement the class `tData` and derived classes to import/allocate and deallocate memory as appropriate. Consider the following cases:

Memory is allocated by `new`. This can be used when the numerical class library will set initial values.

Data is in memory allocated by a visualization system. In this case, the data is memory resident but has been allocated by a visualization system, not by the numerical class library.

The data reside is a file, but it is not necessary to read the entire file into memory. In this case, memory mapping is used.

A portion of the data in an existing `tData` is needed. This occurs when taking a subset of an array or when a file is memory mapped and there is header information such as array dimensions.

When data should not be deallocated by the numerical library under any circumstances. This can occur when you’re just plain paranoid.

`tData` is defined[1]:

```
// Class that keeps reference counts of blocks data for memory
// management. When using these classes, always allocate with new
// and never use the destructors (they’re protected anyway).
// Use ->unreference() when you’re done with the data

class tData
{
private:
     int ReferenceCount;

protected:
     int Bytes; // size in bytes, set by derived classes
     char *Data; // set and (de)allocated by derived classes
     tData() { ReferenceCount = 1; }
     virtual ~tData() = 0;
public:
     char* data() { return Data; }
     void reference() { ReferenceCount++; }
     void unreference() {if ( !(--ReferenceCount) ) delete this;}
     int bytes() { return Bytes; }
     int referenceCount() { return ReferenceCount; }
};
```

When memory allocated by `new` is desired, use:

```
class tNewData : public tData
{
public:
     tNewData( int size ){ Bytes = size; Data = new char[Bytes];;}
protected:
     ~tNewData() { delete[] Data; }
};
```

When the data is in a data structure controlled by visualization system, use:

```
class tVisualizationSystemData : public tData
{
public:
     // tell visualization system this data is referenced
     // and pull out a pointer to the actual data
     tVisualizationSystemData( VisSystemDataType* )[2];
protected:
     ~tVisualizationSystemData();
};
```

1. The code fragments here are modified for clarity: e.g., error checks have been removed.
2. Code here is visualization system dependent.

When the data is in a file that must be memory mapped:

```
class tFileData : public tData
{
public:
     tFileData( const char* filename )[1];
     {
          int fd = open((const char *)filename, O_RDONLY);
          struct stat statbuf;
          fstat(fd, &statbuf);
          Bytes = (int)statbuf.st_size;
          Data = (char*)mmap(0, (int)Bytes, PROT_READ,
                                   MAP_SHARED, fd, 0);
          close( fd );
     }


protected:
     ~tFileData() { munmap( (caddr_t)Data, (int)Bytes ); }
};
```

Since a file of data frequently contains header information, there is a need to access the part of a file containing the data. To meet this need use the following:

```
class tPartOfData : public tData
{
protected:
     tData *ReferencedData;
public:
     tPartOfData( tData* ref, int size, int start )
     {
          ReferencedData = ref;
          Bytes = size;
          Data = ref->data() + start;
          ref->reference();
     }
protected:
     ~tPartOfData() { ReferencedData->unreference(); }
};
```

Sometimes there are blocks of data that should never be deallocated. For this case use the following:

```
class tDontTouchData : public tData
{
public:
     tDontTouchData( int size, char* data )
          { Bytes = size; Data = data; }
protected:
     ~tDontTouchData() {}
};
```

1. Note: #include files left out for brevity.

To see how the `tData` class hierarchy is used, consider vector operation on data in a file:

```
{
tData* d = new tFileData( “foo” ); // reference count = 1
tVector v( d );// count = 2, see below
d->unreference(); // count goes to 1
float f = v.product(); // multiply vector elements together
} // when v is destructed, d’s reference count goes to 0
 // so d is also destroyed
```

The `tVector` constructor and destructor might be implemented as follows:

```
class tVector
{
     float* FloatData; // cache a raw pointer to vector
     tData* Data; // tData that manages the vector’s data
     int Length; // length of the vector
public:
     tVector( tData* d )
     {
          Data = d;
          Data->reference();
          Length = d->bytes() / sizeof(float);
          FloatData = (float*)(d->data());
     }

     ~tVector() { Data->unreference(); }
};
```

IV. Future Work

The `tData` class hierarchy works well for blocks of data in virtual memory; but if the data is in memory on a remote machine, the present `tData` cannot help. However, if virtual `operator[]` and `operator=` were implemented, then a class derived from `tData` should be able to get control when the data is accessed and handle the network access transparently (except for performance!).

V. Conclusion

To be useful to many scientific visualization application programmers, C++ class libraries must access data allocated outside the library without making copies. This can be accomplished by a class hierarchy where the derived classes handle application specific allocation and deallocation.

We recommend that numerical class library developers provide a way for their classes to operate on memory allocated by application programmers, perhaps using a mechanism similar to the one presented here.

VI. Acknowledgments

D. Asimov, C. Levit, E. Raible, J. Krystynak, K. McCabe, and B. Klein reviewed this paper and made many excellent comments and corrections. I would like to thank the Applied Research

Branch of the NAS Division of NASA Ames Research Center for the superb working environment I have become addicted to. This work is supported through NASA contract NAS2-12961.

VII. References

[Atwood92] C. A. Atwood and W. R. Van Dalsem, “Flowfield Simulation about the SOFIA Airborne Observatory,” *30th AIAA Aerospace Sciences Meeting and Exhibit*, January 1992, Reno, Nevada, AIAA 92-0656.

[Bancroft90] G. Bancroft, F. Merritt, T. Plessel, P. Kelaita, R. McCabe, and A. Globus, “FAST: A Multi-Processing Environment for Visualization of CFD,” *Proceedings Visualization ‘90*, IEEE Computer Society, San Francisco, 1990.

[Dyad92] Manuals and personal communication, Dyad Software.

[Globus92] A. Globus, “A Software Model for Visualization of Time Dependent 3-D Computational Fluid Dynamics Results,” NASA Ames Research Center, NAS Systems Division, Applied Research Branch technical report RNR-92-031, November 1992.

[Globus93] A. Globus, “Optimizing Time Dependent Particle Tracing,” NASA Ames Research Center, NAS Systems Division, Applied Research Branch technical report RNR-93-0??, 1993. In progress, title subject to change.

[Levit92] C. Levit, personal communication.

[RW92] Manuals and personal communication, Rogue Wave.

[SGI92a] *IRIS Explorer User’s Guide*, Silicon Graphics Computer Systems, Document 007-1371-010, 1992.

[SGI92b] *IRIS Explorer Module Writer’s Guide*, Silicon Graphics Computer Systems, Document 007-1369-010, 1992.

[UNIX] *UNIX* mmap *Man Page*.

[Upson89] C. Upson, et. al., “The Application Visualization System: A Computational Environment for Scientific Visualization,” *IEEE Computer Graphics and Applications*, July 1989, Vol. 9, No. 4, pp 30-42.